Relazione Tirocinio Breve

Realizzazione Infrastrutture per Accesso ai Servizi tramite Carte Multifunzione

Corso di Laurea Triennale in Ingegneria Informatica

Laureando:
Nicola Bertolini

Relatore:
Chiar.mo Prof. Federico Filira

24 Febbraio 2011
Anno Accademico 2010-2011

Ringraziamenti

È sempre difficile fare i ringraziamenti, si ha sempre paura di dimenticare qualcuno o di esser inappropriato. Ma ho scelto ingegneria perché le cose facili non mi piacciono. E allora forse è giusto iniziare dal nonno Silvano che mi ha fatto conoscere il primo computer e la masetra Gabriella che mi ha fatto conoscere la matematica. È gran parte colpa loro se sono diventato così!

Molti sono stati quelli che mi hanno accompagnato in questo viaggio lungo la conoscenza, ne è rimasto uno sparuto gruppetto: Ale , Ser, Lele e Skizzo i compagni perfetti per la colazione all'intermezzo,una briscola al paladei anche con -10, una partita a calcetto (ho detto calcetto?!?) e ogni tanto andare a lezione. Menzione speciale per il mio Team Leader, il Pancio, che ho abbandonato nei meandri di PariPari.

Mi piace ricordare anche tutti i miei insegnati, forse non proprio tutti, forse, effettivamente, pochi di loro, che con passione e molta pazienza mi hanno sopportato. Doveroso il ringraziamento al professor Filira per la sua disponibilità, e a tutti i colleghi di Telerete.

E poi gli amici di tante avventure, campi, riunioni infinite (Andiamo avanti!?!). Alessia perché sei la mia alter ego. Mattia, Alvise e Teresa perché ne avrete le palle piene di sopportare me e i miei casini. Enrico, Valentina e Alessia perché l'esempio è sempre il miglior maestro. E poi Cecilia, Marta, Marco, Andrea, Sara, Francesca e Saretta, perché so di non essere una persona facile con cui lavorare.

Quasi dimenticavo il santo uomo (provate voi a sopportare la Baghi) Manuel che mi ha portato a giocare a rugby.

A Chiara, mia prima compagna di giochi, primo riferimento nella vita. E a Gianluca che la sopporta. E come lei voglio ringraziare Ole Kirk Christiansen inventore del LEGO perché ingegneri si diventa fin da piccoli.

A mamma e papà per gli sforzi fatti per farmi studiare e per il loro appoggio mai mancato in questi anni, e perché è sopratutto colpa loro se oggi siamo qui.

Indice

Premessa

Da aprile 2010 APS Mobilità, il ramo di attività di Aps Holding destinato alla gestione del sistema del trasporto pubblico, ha iniziato la distribuzione delle tessere elettroniche PadovaPass.
L'obbiettivo primario di questa iniziativa è quello di sostituire le vecchie tessere di riconoscimento per i trasporti pubblici con le nuove tessere contactless entro la fine dell'anno per poi lanciare il nuovo sistema di pagamento del servizio pubblico di trasporto tramite le suddette carte. In esse infatti si potrà caricare uno o più titoli di viaggio: dal semplice biglietto ordinario, al carnet, agli abbonamenti per ogni categoria di utente e durata, fino a possibili titoli a scalare a consumo. Ciò comporta anche una distribuzione presso i punti di rivendita autorizzati di dispositivi specifici per caricare nelle tessere i titoli di viaggio desiderati. Una volta distribuite, le tessere, saranno in possesso della maggior parte della popolazione che sfrutta i servizi forniti dal Comune di Padova e da APS Holding.

In questo contesto la tessera PadovaPass sarà utilizzabile anche per gestire tariffe integrate con le altre aziende di trasporto, o per l'acquisto di altre tipologie di servizi (ingresso a musei, spettacoli, pagamento di bollette, accesso e prestito presso le biblioteche comunali, ecc..) con la possibilità di diventare una vera e propria Carta del Cittadino. Qui si inserisce l'analisi che verrà svolta in quest'elaborato. Si vuole capire effettivamente quali servizi potranno essere forniti alla cittadinanza tramite le nuove tessere e come questa offerta potrà essere realizzata. A questo fine si discuterà come PadovaPass potrà essere utilizzata per il servizio di prestito di libri presso le biblioteche comunali e come questa funzione potrà essere integrata nei sistemi di questo genere già esistenti. Tale sistema verrà, infine, realizzato.

Stato dell'arte

Allo stato attuale sono in distribuzione le tessere PadovaPass e, dato che per il rinnovo degli abbonamenti è obbligatorio acquistare la nuova tessera, la distribuzione a chi afferisce ai servizi di trasporto è ormai ultimata. Queste tessere contactless — senza contatto — appartengono alla famiglia delle smartcard MiFare della NXP Semiconductors e le caratteristiche specifiche verranno trattate in seguito. Basti sapere che queste tessere dispongono di

una certa quantità di memoria libera, che ogni tessera ha un suo ID univoco e che sono compatibili con lo standard ISO-14443 a 13.56 MHz.

Inoltre abbiamo a disposizione un sistema bibliotecario, realizzato da @bc network s.a.s., che utilizza una diversa tecnologia RFID conforme allo standard ISO-15693 a 13.56MHz.

Dove vogliamo arrivare?

Si vuole riuscire a sfruttare il sistema già esistente e integrarlo con le tessere MiFare PadovaPass in modo che per accedere al servizio bibliotecario non sia necessario possedere un'altra, l'ennesima, carta.

Rotta da seguire

Analizzeremo come sfruttare le tessere MiFare e quali funzionalità implementate in esse ci saranno utili. Poi, partendo dal codice in Delphi .NET e il sistema funzionante sviluppato da @bc network, si studierà come funziona tale sistema e si punterà a sfruttare il più possibile ciò che è già realizzato per integrare nel sistema la tessera PadovaPass

Capitolo 1

Sistemi RFID

RFID, acronimo che sta per Radio Frequency IDentification, è una tecnologia che utilizza la comunicazione via onde radio per scambiare dati tra un lettore e un dispositivo elettronico detto Tag. RFID permette quindi di memorizzare e accedere a distanza a dati memorizzati sui Tag i quali sono in grado di rispondere comunicando le informazioni in essi contenute quando interrogati.

Sotto il nome di RFID si nascondono diverse tecnologie descritte da diversi protocolli di comunicazione. In particolare si differenziano fra di loro per le frequenze su cui lavorano, le distanze a cui riescono a comunicare e la funzione che svolgono.
In questo elaborato lavoreremo con sistemi RFID che si appoggiano a onde radio con frequenza 13.56 MHz. Fra le tecnologie che utilizzano tale frequenza portante analizzeremo due protocolli: ISO-14443 e ISO-15693

1.1 Il sistema RFID

Un sistema RFID è composto da un apparecchio di lettura (lettore RFID) e dai Tag RFID.

Un Tag è composto sempre da almeno due parti. Una è un circuito integrato per l'archiviazione e l'elaborazione delle informazioni, in grado di modulare e demodulare un segnale radio e compiere altre operazioni di calcolo specializzate più o meno complesse. Tra le informazioni contenute nel microchip è presente anche un identificatore univoco (UID) scritto nel silicio. La seconda è un'antenna per la ricezione e l'invio dei segnali.

Il lettore deve essere in grado di interpretare i dati ricevuti attraverso la propria antenna. In base alla funzione che il sistema deve implementare, sarà necessaria o meno la connessione, diretta o indiretta, del sistema ad un calcolatore.

1.1.1 I Tag RFID

Esistono 3 diversi tipi di Tag che si diversificano per alimentazione.

Attivi sfruttano una batteria per comunicare una volta identificati dal lettore;

Passivi non hanno alcuna fonte di alimentazione e richiedono un campo elettromagnetico esterno per avviare una trasmissione;

Passivi assistiti da batteria devono essere risvegliati da una fonte esterna ma poi utilizzano la batteria interna per trasmettere a maggior potenza.

I Tag da 13,56 MHz sono previsti dalle norme ISO come passivi (senza batterie).

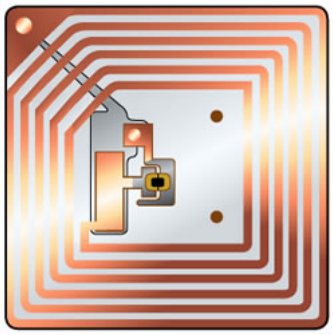

Figura 1.1: Un Tag RFID

I Tag passivi

Il funzionamento del sistema passivo sfrutta il principio di induzione elettromagnetica. Il Lettore eccita l'antenna con un segnale ad una specifica frequenza radio. L'onda elettromagnetica viene trasformata, per il principio dell'induzione, in corrente elettrica che andrà ad alimentare il chip che compirà le semplici operazioni, d'accesso o scrittura, richieste e poi trasmetterà i dati attraverso l'antenna.

1.1.2 Vantaggi rispetto altre tecnologie

Questa tecnologia ha sostanziali vantaggi rispetto alle tradizionali tecnologie a codici a barre o a banda magnetica.

- Non deve essere a contatto per essere letto come per le bande magnetiche;

- Non deve essere visibile per essere letto come per i codici a barre;
- Si possono anche aggiungere informazioni sui chip in funzione della tipologia dello stesso;
    1. Read Only: si possono solo leggere le informazioni contenute;
    2. Write Once, Read Many: si possono scrivere nel chip le informazioni una sola volta, ma leggerle; un numero illimitato di volte;
    3. Read and Write: si possono leggere e memorizzare informazioni per un numero limitato ma grande di volte.
- L'identificazione e la verifica avvengono in un decimo di secondo;
- La comunicazione può essere in chiaro o cifrata.

Avendo, mediamente, maggior capacità rispetto ad una banda magnetica o un codice a barre, possono subentrare in sistemi già esistenti che utilizzano le vecchie tecnologie affiancandosi ad esse, rendendo così accessorio un aggiornamento completo del sistema coinvolto.

1.1.3 Integrazione

I Tag RFID, grazie alle loro ridotte dimensioni, possono essere integrate in praticamente qualsiasi oggetto. Molto diffuse sono le etichette per la tracciatura: sul retro presentano il circuito RFID che si incolla direttamente sull'oggetto desiderato, mentre sul fronte sono stampabili e quindi possono essere accompagnate da codici a barre con le medesime informazioni.
Molto diffuse sono anche le tessere tipo carta di credito o i portachiavi, entrambi più adatti al controllo d'accesso.

Figura 1.2: Due TAG RFID integrati: in un etichetta stampata e in un portachiavi

1.1.4 Applicazioni

La tecnologia RFID ha trovato applicazione in diversi campi tra i quali: Passaporti, Monetica, **Bigliettazione Elettronica(PadovaPass nasce per questo)**, Logistica, Controllo accessi, Tracciamento, Identificazione degli animali, Biblioteche, Antitaccheggio.

1.2 Protocolli ISO-15693 e ISO-14443

Sono due protocolli per sistemi RFID a 13,56 MHz. Insieme definiscono la famiglia di smartcard dette Proximity Cards, tessere in grado di comunicare con il lettore a distanze non eccessivamente elevate.

Il protocollo è uno standard e non una normativa, quindi rispettare il protocollo più o meno completamente è una libera scelta dello sviluppatore. I Dispositivi che rispettano il protocollo sono in grado di comunicare completamente fra loro.

1.2.1 ISO-15693

ISO-15693 è uno standard per smartcard appartenenti alla famglia Proximity Card dette *Vicinity Cards*, ovvero le carte che possono essere lette e scritte da distanze superiori al metro, anche fino a 1,5 m.
Sono carte poco potenti dal punto di vista computazionale usate soprattutto per data storage. In particolare sono dotate di una memoria a blocchi a cui si può accedere e scrivere, oltre all'identificatore univoco di sola lettura. Alcuni tipi di tessere sono munite anche di settori di memoria che si possono scrivere più volte ma che con specifichi comandi diventano di sola lettura. È il caso dei campi AFI (Application Family Identifier) che contiene informazioni sul tipo di applicazione a cui è destinata la carta, e DSFID (Data Storage Format Identifier) che rappresenta il formato dei dati salvati nei blocchi di memoria.

I lettori che rispettano lo standard devono essere in grado di gestire più Tag contemporaneamente. Per lo scambio informazione fra il lettore e i Tag sono disponibili 3 modalità:

Addressed Mode (ADD) ad ogni comando viene associato l'UID del Tag che deve eseguire il comando;

Selected Mode (SEL) prima viene selezionato un Tag e poi ogni comando è destinato a quel Tag;

Non-Addressed Mode (NAD) il comando viene mandato in Broadcast a tutti i Tag nell'area attiva.

I comandi che possono essere inviati al Tag sono riassunti in tabella 1.1

Comando	**Modalità**	**Descrizione**
Inventory	NAD	Controlla l'esistenza di Tag ISO-15693 nell'area attiva e restituisce alcune informazioni fra le quali gli UID
Stay Quiet	ADD	Mette il Tag corrispondente in stato di quiete
Lock multiple Blocks	ADD, SEL	Rende immodificabile la serie di blocchi di memoria
Read multiple Blocks	ADD, SEL	Legge una serie di blocchi di memoria del Tag corrispondente
Write multiple Blocks	ADD, SEL	Scrive una serie di blocchi di memoria del Tag corrispondente
Select	ADD	Seleziona un Tag
Reset to Ready	ADD, NAD	Sveglia i Tag messi in stato di quiete
Write AFI	ADD, SEL	Scrive un nuovo valore AFI
Lock AFI	ADD, SEL	Rende immodificabile il valore AFI
Write DSFID	ADD, SEL	Scrive un nuovo valore DSFID
Lock DSFID	ADD, SEL	Rende immodificabile il valore DSFID
Get System Information	ADD, SEL	Ottiene informazioni dettagliate del Tag

Tabella 1.1: Lista dei comandi ISO-15693

1.2.2 ISO-14443

ISO-14443 definisce lo standard per contactless smartcard appartenenti alla famiglia delle Proximity Card. Con il termine *contactless smartcard* indichiamo un genere di tessere multifunzione di prossimità che funzionano solo se posizionate molto vicino al lettore a distanze non superiori ai 10 cm.

Sono tessere in genere più potenti delle vicinity cards, ne esistono tipi con veri processori general purpose che si avvalgono di Sistemi Operativi integrati. Sono anche dotate di sistemi di sicurezza e crittografia dei dati più o meno potenti ed affidabili. In questo genere di tessere esistono due tipi di dato: R/W Block, per immagazzinare 16 byte di dati generalizzati, e il Value-Block, usato per operazioni consecutive di incremento, decremento e ripristino di valori. La dimensione massima di un Value-Block è di 4 byte, ma occupa lo stesso un intero blocco da 16 byte, replicando 3 volte il valore per preservarlo da errori, e riservando, in parte, 4 byte come bit di controllo. Un dato di tipo Value-Block può risiedere in qualsiasi blocco di memoria ma va precedentemente inizializzato.

Il protocollo si divide e viene descritto in 4 parti.

1. Caratteristiche Fisiche;
2. Radiofrequenza e Modulazione;
    - Modalità A;
    - Modalità B.
3. Inizializzazione e Anticollisione;
4. Protocollo di Trasmissione.

La terza e la quarta parte sono da considerare di grosso interesse. Anche ISO-14443 gestisce Tag multipli nell'area attiva e conserva i tre tipi di modalità di comunicazione. Essendo l'area attiva molto ridotta difficilmente più di un Tag comunicherà con l'antenna e verrà riconosciuto dal lettore.

I comandi che possono essere inviati al Tag sono riassunti in tabella 1.2

Le più diffuse carte di questo tipo sono le tessere MiFare.

1.3 MiFare

Come già accennato, MiFare è una tecnologia proprietaria brevettata da NXP Semiconductors basata sullo standard ISO-14443 tipo A (non tutti i tipi di tessera lo implementano completamente). Esse sono di fatto dispositivi di memoria forniti di un identificatore univoco con un certo livello di sicurezza che varia da modello a modello.

La tecnologia proprietaria MiFare è implementata nelle smartcard quanto nei lettori, quindi se si vuole lavorare con tutta la tecnologia bisognerà

Comando	**Modalità**	**Descrizione**
Inventory	NAD	Controlla l'esistenza di Tag ISO-14443 nell'area attiva e restituisce alcune informazioni fra le quali gli UID
Select	ADD	Seleziona un Tag
Authentication	SEL	Autenticazione tramite chiave
Halt	SEL	Il Tag selzionato viene messo in stato HALT
Read multiple Blocks	ADD, SEL	Legge una serie di blocchi di dati dalla carta (16 byte l'uno)
Write multiple Blocks	ADD, SEL	Scrive una serie di blocchi di dati sulla carta (16 byte l'uno)
InitValue	SEL	Inizializza un blocco di memoria come Value-Block
ReadValue	SEL	Legge il valore di un Value-Block
Increment	SEL	Somma il valore passato con quello di un Value-Block e salva il risultato nel registro interno
Decrement	SEL	Sottrae al Value-Block il valore passato e salva il risultato nel registro interno
Restore	SEL	Legge un ValueBlock, controlla che non ci siano errori e salva il valore nel registro interno
Transfer	SEL	Trasferisce il valore del registro interno al Value-Block

Tabella 1.2: Lista dei comandi ISO-14443

realizzare un ricevitore coerente con la tecnologia. Con il nome MiFare in realtà si identificano 7 differenti tipi di carte con diversa capacità di memorizzare dati, diversi tipi di crittografia e differente potenza di calcolo. Nella nostra analisi focalizzeremo l'attenzione sui modelli di tessere MiFare distribuite o in prossima distribuzione come PadovaPass: MiFare Classic e, in futuro, MiFare Plus che sono state studiate dal produttore per sostituire le classic.

1.3.1 Classic

Le carte MiFare Classic sono implementate con circuiti integrati per applicazione specifica ed hanno ridotta potenza di calcolo, così divengono, essenzialmente, un dispositivo di memoria, ove lo spazio è diviso in segmenti e blocchi protetti da semplici meccanismi di controllo degli accessi.
Ne esistono, a loro volta, di dimensioni diverse. Le MiFare Classic 1K dispongo di 1024 byte di memoria suddivisi in 16 segmenti. Ogni settore è protetto da 2 chiavi che vegono chiamate A e B. Le MiFare Classic 4K offrono 4096 byte divisi in 40 segmenti, di cui 32 sono delle stesse dimensioni dell'1K, mentre i restanti 8 hanno dimensioni quadruple. Le Mini MiFare Classic offrono 320 byte divisi in 5 segmenti. Per tutti i tipi di carta, 16 byte a settore sono riservati alle chiavi e alle condizioni d'accesso e non possono essere utilizzati per i dati utente. Inoltre i primi 16 byte della tessera contengono il numero di serie univoco e altri dati riguardanti il produttore. Questi dati sono scritti nel silicio e quindi di sola lettura.
In questo modo le capacità delle tessere si riducono a 752 byte per le Classic 1k, 3440 byte per le Classic 4k, e 224 byte per le Mini. In questo particolare tipo di tessere MiFare si utilizzano le prime 3 parti del protocollo ISO-14443(caratteristiche fisiche, potenza del segnale radio e interfaccia di segnale, inizializzazione e anticollisione), mentre un protocollo proprietario sostituisce la quarta parte: trasmissione dati. L'esistenza di un diverso protocollo proprietario non permette in alcun modo l'accesso tramite lettori e funzioni standard del protocollo ISO-14443.

Il basso costo, la semplicità e l'affidabilità di queste carte hanno fatto in modo che avessero un grande successo e fossero impiegate nelle più disparate applicazioni

Sicurezza

Classic implementa l'algoritmo di crittografia Crypto-1 creato appositamente da NXP semiconductors per queste tessere. Il funzionamento di tale algoritmo non interessa, basti sapere che nel dicembre 2007 un gruppo di ricerca ha dichiarato di essere riuscito a superare l'algoritmo di cifratura delle carte Classic. Un analogo risultato è stato ottenuto nel marzo 2008 da un altro

gruppo di ricerca. NXP ha confermato e riconosciuto il problema, che esiste solo per le carte Classic.

1.3.2 Plus

MiFare Plus nasce appositamente per sostituire Classic ed è quindi compatibile con esse per la gestione dei dati e delle applicazioni. Plus fornisce infatti solo un aggiornamento delle infrastrutture di sicurezza di Classic che obbligano però ad un aggiornamento anche dei lettori. Esistono carte da 2 o 4 KB. Per le carte da 4KB sono disponibili modelli con identificativo universale sia da 4 byte, come per le classic, sia da 7 byte. Per le tessere da 2Kb esiste solo la versione con UID da 7 byte.

Sicurezza

Plus supporta AES a 128 bit il che lo rende molto più sicuro rispetto a classic dato che sono stati sviluppati solo attacchi a forza bruta per forzare AES e l'unico conosciuto che sia andato a buon fine ha impiegato 5 anni su una chiave a 64 bit utilizzando il tempo libero di migliaia di CPU di volontari.

Capitolo 2

I Sistemi Obid®

Per lo sviluppo di questo progetto sono sono state messi a disposizione dalla @bc network s.a.s. dei sistemi RFID Obid®della ditta Feig. Questi sistemi sono stati pensati dal produttore per essere personalizzati dai programmatori intermedi al fine di creare un canale di comunicazione a più livelli fra un computer e un Tag.
Tali sistemi sono composti da un'ampia scelta di hardware, antenne e lettori, e una parte software composta essenzialmente da una Software Development Kit(SDK) multipiattaforma. Vengono inoltre forniti alcuni programmi di debug e dimostrazione.

2.1 Sistemi a disposizione

Obid® è, quindi, una famiglia di prodotti molto vasta che copre soluzioni per la quasi totalità delle tecnologie RFID, al variare della frequenza portante e del protocollo di comunicazione.

Classic (LF) per applicazioni a bassa frequenza (Low Frequency – 125kHz);

Classic-pro (HF) per applicazioni ad alta frequenza (High Frequency – 13.56MHz) pienamente compatibili con ISO-14443 e ISO-15693;

i-scan HF per applicazioni ad alta frequenza pienamente compatibili con ISO-15693 e ISO-18000-3;

i-scan UHF per applicazioni ad altissima frequenza (Ultra High Frequency – 865 - 960 MHz);

megalock per sistemi di apertura porte elettronica a bassa frequenza.

@bc network ci ha messo a disposizione diversi tipi di lettore. In particolar modo, il sistema Biblioteca, che verrà presentato nel capitolo 5, è integrato con un lettore **ID ISC.MR101** della famglia **i-scan**. Visto che

questo tipo di lettore non gestisce i Tag di tipo ISO-14443, verrà presentato anche il lettore **ID CPR40.30** della famiglia **Classic-pro**

ID ISC.MR101 è un lettore per vicinity card, per questo deve disporre anche di un antenna delle dimensioni e della potenza adeguate per comunicare con Tag a distanze relativamente elevate. Per questo il lettore non integra direttamente l'antenna: essa si collega al lettore tramite un cavo coassiale. Per i diversi tipi di applicazione FEIG mette a disposizione varie antenne, a noi è stata fornita l'antenna **ID ISC.ANT340/240**. Il lettore, alimentato da rete, si collega al PC tramite un cavo USB 1.0 / USB 2.0 per questo è necessario installare i driver adatti al lettore.

ID CPR40.30 è invece un lettore per proximity card e quindi richiede che la carta sia molto vicina all'antenna integrata nel lettore. Si collega al PC tramite cavo miniUSB / USB 2.0 dal quale trae anche l'alimentazione necessaria al funzionamento. Dispone anch'esso dei suoi driver specifici.

2.2 la connesione a Obid®

Come già accennato Obid® mette a disposizione una SDK multipiattaforma. Essa è composta da un insieme di librerie di sviluppo, librerie dinamiche (DLL) e driver che rende possibile lo sviluppo di software su varie piattaforme con diversi linguaggi di programmazione per differenti prodotti Obid®.

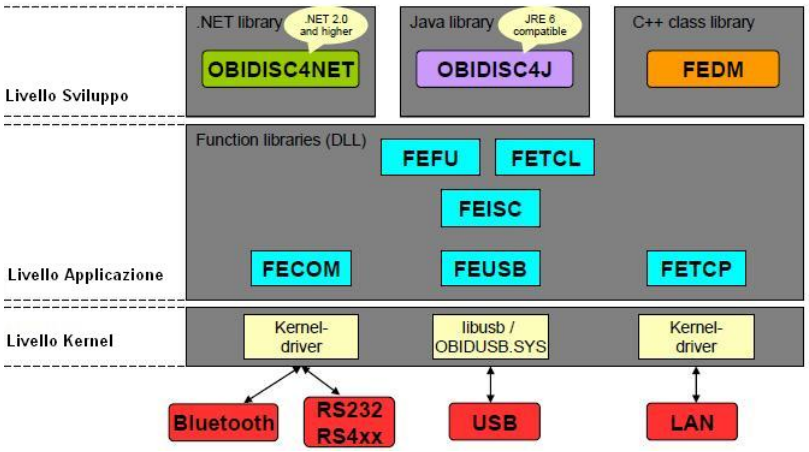

Figura 2.1: Schema riassuntivo della SDK Obid®

Fra i Sistemi Operativi supportati (non solo per lo sviluppo ma anche per l'utilizzo applicativo) troviamo Windows 2000/XP/Vista(32-bit), Windows Mobile e Linux(x86). Come mostrato in figura 2.1, le piattaforme di sviluppo supportate sono C++, .NET e Java.

2.2.1 Livello Kernel

FEIG mette a disposizione diverse modalità di comunicazione seriale dei sistemi Obid® con il computer. In particolare fornisce dispositivi con interfaccia per la porta seriale (RS232 o RS4xx), con interfaccia USB 2.0 e con interfaccia TCP/IP. A livello Kernel viene posto l'insieme dei driver che permette al sistema operativo di comunicare con i lettori attraverso le varie interfacce. Non permette però la comunicazione diretta delle applicazioni.

Dovendo utilizzare più apparati ci si è scontrati con l'esigenza di installare driver per diversi dispositivi. In particolar modo vengono fornite diverse versioni del file di sistema **obidusb.sys** che non possono essere installate contemporaneamente nella cartella **C:\Windows\System32\Drivers**. Il problema è esteso dal fatto che per alcuni dispositivi i driver da installare non si limitano al file di sistema sopraccitato ma sono accompagnati da altri file. Dopo numerosi tentavi di trovare una compatibilità si è scelto di installare i driver in ordine cronologico inverso, partendo, cioè, dai driver del dispositivo con la versione di obidusb.sys più recente, a quella meno recente. In questo modo si è riusciti a far funzionare tutti i dispositivi utili insieme. La versione di obidusb.sys installata in questo modo risulta essere la 2.40.

Si è però riscontrata anche una generale instabilità dei driver, forse dovuta al fatto che non si installa la versione più recente. Spesso durante lo sviluppo del software i dispositivi hanno smesso di funzionare o di essere riconosciuti dal PC dopo un non meglio determinato periodo di tempo nel quale sono rimasti inutilizzati, ed è anche capitato di dover reinstallare i driver o per lo meno riavviare il PC per riottenere il controllo dei dispositivi. In generale si è riscontrato che non è consigliabile aprire e chiudere connessioni con i dispositivi in continuazione, ma è meglio, una volta aperta la comunicazione, mantenerla aperta per tutta la sessione di lavoro.

2.2.2 Livello Applicazione

Per comunicare con i dispositivi, le applicazioni devono passare attraverso delle funzioni standard implementate in alcune librerie dinamiche. Tali librerie adempiono a due diversi compiti e quindi possono essere divisi in altri due livelli virtuali.
Il primo di essi, composto dalle DLL FECOM, FEUSB e FETCP, rispettivamente per dispositivi con collegamento COM, USB o TCP/IP, si occupa di rendere uniforme la vista, da parte delle altre librerie, delle diverse interfacce attraverso funzioni di scansione delle porte, apertura di canali di comunicazione, scambio dati e diagnosi errori.
Il secondo, composto dalle DLL FEFU, FETCL e FEISC o dalle loro varianti per Windows Mobile, dispone di tutte quelle funzioni per la comunicazione secondo i protocolli ISO. Esse però sono specifiche del dispositivo ovvero

del protocollo o dei protocolli supportati. Per un'ulteriore approfondimento rimandiamo alla sezione 2.3

2.2.3 Livello Sviluppo

A Questo livello ci sono le librerie che permettono di programmare secondo una filosofia orientata agli oggetti con i diversi linguaggi senza dover per forza conoscere nello specifico le funzioni e le funzionalità delle DLL del livello applicazione.
Resta al programmatore la scelta di utilizzarle o meno, in virtù della funzione e della destinazione del software e del codice.

Esiste una libreria per ogni piattaforma sopraelencata: FEDM per c++, OBIDISC4NET per .NET, progettata per .NET 2.0 o superiori, e OBIDISC4J per Java, compatibile con JRE 6.

2.3 Le librerie della SDK

Nel nostro caso non faremo uso di funzioni a livello Sviluppo ma ci appoggeremo a delle unit di delphi che richiamano direttamente le funzioni delle DLL. Per questo motivo è importante conoscere nello specifico le funzioni di cui esse dipongono e il loro funzionamento.
Non tutte le librerie a livello Applicazione sono utili al nostro scopo, in particolare ci interessa la connessione usb che risiede in FEUSB e le funzioni di comunicazione che si trovano in FEISC.

2.3.1 Le DLL

Una dynamic-link library è una libreria software che viene caricata dinamicamente in fase di esecuzione, invece di essere collegata staticamente ad un eseguibile in fase di compilazione. La separazione del codice in librerie (dinamiche o statiche) permette di spezzare i programmi in parti concettualmente separate. Il principale vantaggio delle librerie a collegamento dinamico è che esse verranno caricate solo se effettivamente necessarie durante l'esecuzione. Inoltre permettono di risparmiare risorse del sistema caricando in memoria ogni libreria una sola volta anche se utilizzata da più task. Questo metodo, di loading on demand, consente, inoltre, installazioni parziali di un sistema software, in cui sono effettivamente presenti sulla memoria di massa solo le librerie associate alle funzioni che l'utente desidera, come nel nostro caso saranno presenti solo FEUSB e FEISC anche se l'SDK è composta da molte altre librerie. Questo approccio a file separati permette un aggiornamento automatico di tutti i programmi che le usano, senza necessità di ricompilarli, tramite la sola modifica delle DLL, a patto che le due versioni siano compatibili. Il principale svantaggio è legato al fatto che una nuova versione di una DLL potrebbe effettuare dei cosiddetti breaking changes in modo

volontario o a causa di bug. Un breaking change è una modifica del comportamento di una funzione che la rende non più compatibile con le convenzioni in uso. Ancora più critico il caso in cui un programma di installazione sovrascriva una DLL con una versione più vecchia. Questi problemi, ben noti ai programmatori Windows, sono chiamati in gergo DLL Hell (inferno delle DLL).

2.3.2 La libreria FEUSB

Come già accennato, questa libreria, da la possibilità di comunicare tramite il driver e la porta USB con un lettore.
I dispositivi USB della famiglia Obid® sono caratterizzati dal fatto che hanno un'interfaccia uniforme. Al programmatore resta da collegare il driver e l'interfaccia ad un'oggetto per gestire la connessione. Per questo, FEUSB, mette a disposizione un modello di programmazione che consente la Comunicazione in 4 semplici passi

1. Scansione: una funzione trova tutti i dispositivi Obid® sulla porta USB e li gestisce in una *scan list* all'interno della DLL;

2. Selezione: la *Scan list* viene utilizzata per selezionare un dispositivo USB in base al suo numero seriale. Il numero seriale è l'unico attributo che distingue i dispositivi l'uno dall'altro;

3. Apertura di un canale di comunicazione: viene aperto un canale USB che viene assegnato ad una struttura dati creata internamente alla DLL;

4. Scambio Dati.

Per fare ciò FEUSB mette a disposizione diverse strade che utilizzano diverse funzioni. È importante sapere che nel caso si volesse lavorare con più dispositivi collegati bisognerebbe utilizzare la struttura dati *Scan list* i cui nodi interni salvano per ogni dispositivo collegato una serie di informazioni: un intero per numero seriale, un intero che rappresenta il Device-handle, una stringa che contiene la descrizione della famiglia a cui appartiene, un'altra stringa che specifica il modello e un flag booleano che indica se il dispositivo in questione è ancora collegato al PC. Per navigare tale lista bisogna avvalersi di alcune funzioni `FEUSB_Scan` per cercare in tutte le interfacce USB virtuali i dispositivi e salvarli nella *Scan list*, `FEUSB_GetDeviceList` per navigarla ciclicamente, `FEUSB_ClearScanList` per cancellarne il contenuto.

Trovati i dispositivi non resta che sceglierne uno o pù e aprirne i canali.

```
int FEUSB_OpenDevice(long nDeviceID)
```

Passandogli un numero seriale di un dispositivo presente nella *Scan list* essa restituisce il Device-handle che verrà poi usato da FEISC per creare il lettore virtuale associato a quello fisico.

Per il trasferimento dati FEUSB mette a disposizione `FEUSB_Transmit` per trasmettere dati dal PC al lettore, `FEUSB_Receive` per mettersi in ascolto sul canale USB e `FEUSB_Transceive` per trasmettere e ascoltare la risposta. È da sottolineare che i dati inviati a questo livello non hanno ne valore di comandi ne di dati veri e propri, sono da intendersi solo come insieme di byte. La comunicazione semantica avverrà tramite FEISC

La più veloce ed immediata maniera per aprire la comunicazione in un singolo dispositivo è costituita da una singola funzione con determinati parametri.

```
int FEUSB_ScanAndOpen(int iScanOpt, FEUSB_SCANSEARCH* pSearchOpt)
```

Per usarla per un singolo dispositivo bisogna passare la costante `FEUSB_SCAN_FISRT` come primo parametro e una struttura vuota (non viene utilizzata) come secondo. Se l'operazione va a buon fine viene restituito il Device-handle associato al dispositivo. Altrimenti restituisce un codice d'errore minore di zero. Se l'opzione di scansione invece fosse `FEUSB_SCAN_ALL`, si aprirebbero canali di comunicazione con tutti i dispositivi connessi all'USB.

A questo punto si può delegare il controllo alla libreria FEISC che gestirà la comunicazione. È importante, in ogni caso ricordarsi di chiudere il canale di comunicazione e liberare la memoria con la funzioneint FEUSB_CloseDevice(int iDevHnd) passandogli il Device-handle come unico argomento.

2.3.3 La libreria FEISC

Questa libreria permette al programmatore di vedere il proprio dispositivo uniformemente rispetto al tipo di collegamento e quindi di comunicare con esso senza dover pensare a gestire il canale, a mettersi in ascolto o trasmettere, alla divisione dei dati da inviare.
Prima però è necessario creare un lettore virtuale internamente alla DLL tramite la

```
int FEISC_NewReader(int iPortHnd)
```

passando il Device-handle ottenuto da FEUSB. Se viene restituito un intero maggiore di 0 allora, questo, è il Reader-handle associato al lettore virtuale, altrimenti è un codice d'errore. Ad ogni lettore viene associato un canale virtuale per la comunicazione seriale indipendente dal tipo di periferica. Per inviare messaggi in broadcast a tutti i lettori si può usare l'indirizzo riservato 255. Per inviare comandi e dati si possono utilizzare due notazioni basate

su array di caratteri. La prima per interpreta ogni carattere come carattere ASCII e l'intero array come vero array, quindi senza sentinella nulla; ad esempio 0x23, 0x56, 0xFA, 0xB3 sarà l'array {'#' , 'V' , '.' , '+'}. La seconda interpreta l'array di caratteri come stringa di C, quindi necessita del carattere nullo come sentinella finale. In questa notazione ogni carattere corrisponde ad una cifra esadecimale quindi due cifre faranno un byte di dati; ad esempio gli stessi dati di prima saranno rappresentati "2356FAB3". In questo caso la lunghezza raddoppia. Tutti gli altri dati contenuti in un singolo carattere vanno interpretati come singolo byte.

Per la comunicazione tramite comandi ci affidiamo ad una delle funzioni speciali per la comunicazione presenti in FEISC. Questo gruppo di funzioni non funzionano su tutti i lettori ma sono molto potenti e semplificano di molto il codice e l'approccio stesso con la comunicazione fra PC e lettore fisico.
In particolare

```
int FEISC_0xB0_ISOCmd( int iReaderHnd, UCHAR cBusAdr, UCHAR* cReqData,
int iReqLen, UCHAR* cRspData,int* iRspLen, int iDataType )
```

è compatibile con tutti i lettori e permette di inviare comandi specifici al lettore e riceverne la risposta con una singola istruzione anche se complessa. Come parametri bisogna fornire il Reader-handle, il numero del canale virtuale, un array di caratteri contenente il comando, la lunghezza del comando, un array di caratteri per contenere la risposta, un puntatore a intero per ricevere la lunghezza della risposta, e un intero che indica il formato del dato inviato e da ricevere. Inoltre la funzione restituisce un intero che vale 0 nel caso l'esecuzione del comando sia avvenuta correttamente, mentre restituisce un intero negativo per segnalare un errore.

Per alcune funzioni specifiche del protocollo è necessario inviare le istruzioni tramite la gemella di `FEISC_0xB0_ISOCmd` che è `FEISC_0xB2_ISOcmd` che nello specifico accetta le istruzioni di autentificazione e gestione dei Value-Block per MiFare.

Per i comandi per i Tag di nostro interesse si rimanda alle sezioni 2.4.1 e 2.4.2

Anche con FEISC bisogna liberare la memoria occupata dall'oggetto interno alla DLL, il Reader-handle, tramite la funzione

```
int FEISC_DeleteReader( int iReaderHnd )
```

2.4 La comunicazione

Ora abbiamo tutti i mezzi per realizzare la comunicazione fra l'applicazione e il Tag. Ripassando gli elementi che intervengono sono, in ordine:

1. Applicazione;
2. FEISC.DLL;
3. FEUSB.DLL;
4. Driver della Porta USB;
5. collegamento fisico per trasmissione seriale;
6. Lettore;
7. Antenna;
8. Tag.

È da stabilire la lingua che va parlata, l'insieme, cioè, dei comandi che l'applicazione può mandare e la struttura delle risposte che il Tag fornisce.

Le risposte sono logicamente dipendenti dai comandi conformemente all'ottica master – slave che è implementata nel sistema. Resta da analizzare quali comandi si possono inviare e quali funzioni possono eseguire i Tag.

2.4.1 Protocollo ISO-15693

I comandi presentati in tabella 1.1 hanno una corrispondenza univoca nell'alfabeto delle istruzioni che si possono far eseguire a un Tag ISO-15693. Resta da analizzare che genere di dati passare tramite `FEISC_0xB0_ISOCmd`, nel campo cReqData, e quali informazioni, e in che formato, ogni istruzione restituisce tramite il campo cRspData.

Nella tabella 2.1 si trovano i codici dei comandi, la lunghezza delle risposte in byte e la forma di queste. Per i comandi con possibilità di modalità Addressed e Selected il campo MOD[1] sarà presente solo nella prima delle due.

2.4.2 Protocollo ISO-14443

I comandi di ISO-14443 si dividono in comandi standard e comandi speciali. I primi, riassunti in tabella 2.2 sono simili a quelli descritti in tabella 2.1 per ISO-15693, e si eseguono sempre con `FEISC_0xB0_ISOCmd`. La seconda categoria è l'insieme dei comandi per accedere ai Value-block e per l'autenticazione quando essa è richiesta dal Tag e utilizza la funzione `FEISC_0xB2_ISOCmd` identica per parametri e valori restituiti ma che impone un diverso protocollo di comunicazione.

Comando	**Forma del comando**	**Forma della Risposta**
Inventory	0x01, MOD[1]	{Type[1], DSFID[1], UID[8]} * NTag
Stay Quiet	0x02, MOD[1], UID[8]	NULL
Lock multiple Blocks	0x22, MOD[1], UID[8], BloccoIniziale[1], NBlocchi[1]	NULL
Read multiple Blocks	0x23, MOD[1], UID[8], BloccoIniziale[1], NBlocchi[1]	{Blocco[4]} * NBlocchi
Write multiple Blocks	0x24, MOD[1], UID[8], BloccoIniziale[1], NBlocchi[1], {Blocco[4]} * NBlocchi	NULL
Select	0x25, MOD[1], UID[8]	NULL
Reset to Ready	0x26, MOD[1], UID[8]	NULL
Write AFI	0x27, MOD[1], UID[8], AFI[1]	NULL
Lock AFI	0x28, MOD[1], UID[8]	NULL
Write DSFID	0x29, MOD[1], UID[8], DSFID[1]	NULL
Lock DSFID	0x2A, MOD[1], UID[8], DSFID[1]	NULL
Get System Information	0x2B, MOD[1], UID[8]	DSFID[1], UID[8], AFI[1], MemorySize[2], ICRefrence[1]

Tabella 2.1: Forma dei comandi e delle risposte ISO-15693

Comando	Forma del comando	Forma della risposta
Inventory	0x01, MOD[1]	{Type[1], Info[1], UID[8]} * NTag
Select	0x25, MOD[1], UID[8]	NULL
Halt	0xC0, MOD[1]	NULL
Read multiple Blocks	0x23, MOD[1], UID[8], BloccoIniziale[1], NBlocchi[1]	{Blocco[16]} * NBlocchi
Write multiple Blocks	0x24, MOD[1], UID[8], BloccoIniziale[1], NBlocchi[1], {Blocco[14]} * NBlocchi	NULL

Tabella 2.2: Lista dei comandi ISO-14443

Il comando di autenticazione richiede di conoscere una delle due chiavi della tessera (Key A o Key B) e che questa sia nota al programmatore o salvata nella memoria EEPROM del lettore. Per questa operazione FEISC ci mette a disposizione

```
int FEISC_0xA2_WriteMifareKeys(int iReaderHnd, UCHAR cBusAdr, UCHAR cType,
UCHAR cAdr, UCHAR* cKey, int iDataType)
```

Dove cType rappresenta la chiave (0 = Key A, 1 = Key B), cAdr è l'indirizzo nella memoria del lettore dove salvare la chiave, e cKey è un array di caratteri dove immagazzinare la chiave da passare; mentre iReaderHnd, cBusAdr e iDataType hanno sempre lo stesso significato. Da qui, se passiamo direttamente la chiave, allora la forma del comando sarà

0xB0, 0x0A, KEYType[1], DBADR[1], KEY[6]

dove DBADR è l'indirizzo del blocco dati da autenticare con la chiave. Passando attraverso la chiave salvata nel reader il comando sarà

0xB0, 0x02, KEYType[1], DBAdr[1], KEYAdr[1]

dove KEYAdr è l'indirizzo nella EEPROM del lettore dove risiede la chiave precedentemente salvata.

Per Eseguire le operazione su ValueBlock disponiamo di un unico comando.

0X30, 0x02, ValCMD[1], SRC[1], OPVal[1], DEST[1]

Dove ValCMD vale 0x01 per l'incremento, 0x02 per il decremento, 0x03 per la copia. A differenza della definizione dei comandi per Value-Block viene sempre richiesto un indirizzo in memoria di destinazione da mettere in DEST e l'inizializzazione avviene in maniera automatica.

Le risposte ai comandi speciali di ISO-14443 non sono state testate in quanto non si conoscono le chiavi dei Tag a disposizione.

Capitolo 3

Possibilità realizzative

Nei capitoli precedenti abbiamo analizzato l'insieme delle conoscenze e delle tecnologie che rientrano nel progetto al quale puntiamo. Ora metteremo a fuoco quali potenzialità possiamo sfruttare, e quali problemi, che abbiamo riscontrato, ci complicheranno il lavoro. È da ricordare il fine ultimo del nostro lavoro, che è la costituzione di un sistema di accesso ai servizi tramite tessere multifunzione, in particolare la possibilità di usare le tessere PadovaPass per integrare sistemi già esistenti e rendere così tale tessera l'unica vera tessera dei servizi al cittadino

3.1 Potenzialità

Le potenzialità di queste tecnologie sono molto ampie e non siamo certo noi a scoprirle. Alcune di esse sono subito evidenti, buona capacità di calcolo, memoria relativamente ampia, sicurezza delle informazioni, univocità dei numeri seriali, ampia quantità di soluzioni per la costituzione dei sistemi di lettura, velocità e semplicità di comunicazione.

Su altre è bene porre maggiore attenzione.

3.1.1 Connessione

Abbiamo parlato di Connessione USB, o tramite altre interfacce seriali. L'aspetto della parola connessione che vogliamo far risaltare in questo paragrafo è necessità o meno di una connessione ad una rete o ad un sistema più complesso del semplice controllore. I sistemi RFID sono pensati per non necessitare la comunicazione con un server, bensì puntano ad essere il più possibile indipendenti, e al più vogliono essere accompagnati da un microcontrollore in grado di mandare i comandi in modalità seriale. Molto utile a questo proposito, e molto comuni nel mercato, sono i dispositivi con interfaccia RS232, obsoleta per la comunicazione con moderni calcolatori, ma ancora molto in voga per i microcontrollori. Si stanno diffondendo anche applicazioni che si

appoggiano a dispositivi mobili quali smartphone o palmari(sopratutto co windows mobile). Questi sistemi vengono usati, ad esempio, da i controllori dei biglietti nei trasporti, o per scaricare saltuariamente i dati salvati in nei Tag.

Grazie alla libertà dalla connessione ad un computer centrale questo genere di tecnologie hanno trovato grande utilizzo in sistemi in movimento, o non raggiungibili da reti estese, quali trasporti, ski-pass, accesso eventi.

3.1.2 Data Storage

È stata sottovalutata anche la locazione dove i dati vengono salvati, essi infatti non vengono salvati nei sistemi di lettura, anche se ciò e possibile, ma richiederebbe la connessione ad un server centrale di tutti i lettori che svolgono tale funzione. I dati vengono salvati nel Tag stesso e protetti dai sistemi di sicurezza e cifratura implementati in esso. In questa maniera i dati viaggiano assieme al Tag e nel periodo nel quale il Tag non è a contatto con un antenna in grado di leggerlo, tali dati sono al sicuro da una varietà di problemi che possono sorgere dal depositarli in un server. Ciò non toglie che si può duplicare i dati per salvaguardarne l'integrità. Inoltre in questo modo i dati sono sempre disponibili insieme al Tag anche in caso di momentanea mancanza di connessione.

3.2 Problemi

Si è già parlato di molti problemi ma bisogna ora analizzarli, valutare quali siano influenti al nostro scopo e quali no, quali si possono risolvere o by-passare e quali ci impediscono di prendere una determinata strada.

3.2.1 Instabilità dei driver

Questo problema è stato risolto o per lo meno by-passato in due semplici passi

1. Installare i driver in ordine cronologico inverso;
2. Operare meno connessioni/disconnessioni dei dispositivi possibili.

Resta però una componente casuale che non si riesce a gestire. Si raccomanda perciò di installare il sistema finito su un computer appena formattato in maniera da limitare i conflitti con altri driver.

3.2.2 Differenza di protcollo

Gestendo ogni tipo di Tag con il suo lettore non si dovrebbero aver problemi (se non i sopraccittati problemi ai driver). Il problema sta nell'univocità del

UID. Dai produttori viene assicurato che l'UID è univoco per ogni Tag fra i Tag dello stesso tipo. Ciò non viene garantito per Tag di protolli diversi, addirittura, fra ISO-14443 e ISO-15693, gli UID hanno lunghezza differente anche se i comandi restituiscono una stringa da 8 byte per entrambi. ISO-14443 ha, infatti, UID da 4 byte o da 7(fra i quali è comunque garantita l'univocità), mentre ISO-15693 ha UID da 8 byte. Il problema non sta di certo nel momento della lettura, sappiamo da che lettore stiamo assumendo i dati e comunque possiamo domandare al Tag informazioni aggiuntive sul tipo. Il problema si verifica se tale UID va salvato all'interno del sistema di lettura. Si possono distinguere quindi i due tipi estendendo l'UID con un ulteriore byte che potrebbe essere uguale a 0x00 per ISO-15693 e a 0xFF per ISO-14443

3.2.3 Accesso a dati protetti

Questo problema riguarda solo le Tessere MiFare, come abbiamo detto non abbiamo accesso alle chiavi di sicurezza per le tessere PadovaPass, inoltre non conosciamo la parte 4 del protocollo proprietario NXP per le MiFare Classic, quindi non possiamo accedere alle aree di memoria coperte da sicurezza. Questo implica che le uniche informazioni che possiamo ottenere sono quelle restituite da i comandi di inventory. Questa è evidentemente la restrizione più pesante sul nostro progetto, in quanto non ci permette di utilizzare i vantaggi presentati nel paragrafo 3.1.2 e di perdere gran parte delle potenzialità delle tessere MiFare. Diventa quindi necessario lavorare con gli UID delle tessere, uniche informazioni utili e accessibili.

L'analisi portata in questi primi capitoli ci porta verso una strada nella quale useremo le tessere MiFare solo per ottenere un identificazione univoca tramite RFID.
Una volta ottenuto l'UID avremo bisogno di una connessione ad un database perché tali UID ottengano un significato all'interno del sistema. In generale quindi non riusciamo a sfruttare le principali potenzialità di MiFare, d'altro canto la scelta della tecnologia non è contemplata nella nostra analisi in quanto è stata operata da APS Holding, e il nostro compito è analizzare come poter sfruttare Mifare per altri scopi.

Capitolo 4

Delphi: Object Pascal

Come abbiamo visto Obid®lascia grande libertà a proposito dello sviluppo del software mettendo a disposizione librerie per più linguaggi di programmazione orientata agli oggetti. In particolare da anche la possibilità di bypassare quello che viene definito livello sviluppo e utilizzare direttamente le DLL tramite linguaggi alternativi quali Visual Basic (senza appoggiarsi a .NET) o Delphi.

@bc network ha sviluppato più applicazioni che sfruttano i sistemi RFID Obid®con Delphi e ci ha fornito la versione delle librerie per sviluppare in C++, VB, e Delphi oltre ad una routine di connessione d'esempio sviluppata con delphi. Di qui, unitamente alla semplicità dell'linguaggio e la facilità d'uso delle DLL, si è scelto di proseguire per la strada già tracciata e di sviluppare software in Delphi.

Con la parola Delphi in realtà ci si riferisce sia ad un linguaggio di programmazione sia ad un ambiente di sviluppo di proprietà della Borland. Molto utilizzato per applicazioni che utilizzano database, è una delle più semplici piattaforme per applicazioni desktop. Il linguaggio di programmazione è stato pensato inizialmente per sviluppare applicazioni sotto win-32 ed era conosciuto come object pascal. Da pascal, storico linguaggio di programmazione, eredita la sintassi, la struttura dei programmi e i tipi nativi. In più viene introdotta la filosofia OOP.

Curiosa è l'origine del nome; inizialmente avrebbe dovuto chiamarsi AppBuilder, nome registrato in anticipo da Novel. Borland si ritrovò così a dover ideare un nuovo nome per il suo prodotto. Uno dei principali obiettivi di questo nuovo ambiente di sviluppo, era il facile interfacciamento con i principali motori database; il più diffuso e conosciuto motore db all'epoca era proprio Oracle e da qui nacque la frase "Se vuoi parlare con l'Oracolo, devi andare a Delphi".

4.1 La Sintassi di Delphi

Ogni linguaggio ha la sua specifica sintassi. Come già detto Delphi eredita la sintassi da Pascal ma una grossa parte viene ampliata grazie all'implementazione della programmazione orientata agli oggetti. Viene da se che una guida esaustiva su delphi necessiterebbe molto più spazio di quello che gli dedicheremo, perciò in questa sezione analizzeremo le caratteristiche fondamentali necessarie a comprendere gli esempi di codice che verranno proposti in seguito.

4.1.1 Le Unit

La Unit è l'unità fondamentale del codice di Delphi: corrisponde con il file .pas ed è divisa in due parti fondamentali: **Interface** e **Implementation**. La prima è destinata a contenere le definizioni di tipi, variabili, dichiarazioni di funzioni e procedure che si vuole siano accessibili da altre unit. La seconda conterrà il codice privato che non si vuole sia accessibile alle altre unit. Una sorta di compressione in un unico file dei file .h e .cpp per C++. Entrambe le sezioni contengono i campi **Uses** sotto i quali si trovano i nomi dei file, privati dell'estensione, da includere al progetto. Molto importante anche il campo **var** dove vengono definite le variabili globali della unit.

4.1.2 I Metodi

Come ogni linguaggio di programmazione ha la possibilità di implementare metodi. In particolar modo essi non possono essere considerati come statici ma devono sempre essere eseguiti da un oggetto di una specifica classe. Si dividono in metodi procedurali e metodi funzionali. I primi sono come metodi C++ che restituiscono tipi **void**, i secondi sono come comunissimi metodi di C++ e possono quindi restituire qualsiasi tipo primitivo od oggetto. La sintassi però si differenzia proprio sul nome e sulla forma.

```
procedure Class.ProcedureName(Type :  Name,...);

function Class.FunctionName(Type :  Name,...)  :  ReturnType;
```

All' interno di ogni metodo si identificano due sezioni, quella dedicata alla definizione delle variabili e quella dedicata al codice da eseguire fisicamente

Codice 4.1: Struttura di un metodo

```
0 Function Class.FunctionName(Type : Name,...);
  var
    //spazio per la definizione delle variabili interne
  begin
    //spazio per il codice effettivamente eseguito
```

```
5   FunctionName := DatoDaRestituire;
end;
```

Dall'esempio di codice 4.1 si evincono due particolarità. Per primo i commenti si operano mettendo // davanti al testo da commentare, se si vuole commentare più di una riga, il testo allora dovrà essere compreso fra **{ }**. Per secondo si vede che per restituire un valore, nel solo caso di function, si utilizza la forma mostrata in riga 5.

4.1.3 I puntatori

Molto importanti sono i puntatori che permettono di gestire al meglio la memoria. La gestione di essi avviene come in C e ciò rende compatibile Delphi con oggetti di C++ eventualmente creati dalle DLL. Per definire un puntatore si utilizza la sintassi `P : ^integer`, una volta definita la variabile è facile gestirla tramite gli operatori ^(dereferenza, * per C) e @(referenza, & per C).

Codice 4.2: Gestione puntatore

```
0 var
    P: ^Integer;
    X: Integer;
  begin
    P := @X;
5   //cambiare il valore in due modi
    X := 10;
    P^ := 20;
```

Si può anche creare lo spazio in memoria ex-novo tramite il comando `New(P);` , dove P è un puntatore, e liberare la memoria con `Dispose(P)`. Infine un puntatore potrà puntare a nil (null), se non ha valore, o meglio memoria, assegnata.

4.1.4 Gli Eventi

Delphi è stato studiato per semplificare la connessione a database ma poì ha subito varie evoluzioni volte a semplificare al massimo la costruzione di GUI. Alla base di ciò, oltre all'ambiente grafico che viene fornito, c'è anche la gestione degli eventi quali l'attivazione di un form, la pressione di un bottone, il cambiamento di stato di un componente ecc. Questi eventi vengono gestiti chiamando, in maniera automatizzata, procedure del form a cui gli elementi grafici si riferiscono dato che gli stessi elementi sono gestiti come attributi della classe form.

Interessanti sono gli eventi per l'attivazione e la chiusura del form. Essi Fungeranno da preambolo e conclusione di qualsiasi operazione che verrà

compiuta dal form o dagli elementi che lo costituiscono da quando è operativo a quando viene chiuso. Generalmente gli eventi vengono chiamati

```
procedure TForm1.FormActivate(Sender: TObject);
```

Per l'attivazione e

```
procedure TForm1.FormClose(Sender: TObject; var CloseAction: TClos-
eAction);
```

per la chiusura.

4.2 Lazarus

Si è scelto di utilizzare un IDE free che ci permettesse di sfruttare, almeno in parte, il codice prodotto, e resoci gentilmente disponibile, da @bc network. Tale codice è stato scritto utilizzando Delphi 7, prodotto di proprietà della Borland e quindi a pagamento. Lazarus è un clone GPL di Borland Delphi che quindi ci ha permesso di analizzare il codice in maniera corretta anche se non permette di compilare i progetti sviluppati con delphi 7 per 2 essenziali motivi:

1. Lazarus non riconosce e gestisce i file e la struttura dei progetti di Delphi 7

2. Lazarus non usa i componenti ADO per la connessione al database ma sfrutta un clone di essi con nomi diversi per oggetti ma sopratutto per nomi delle funzioni e degli attributi

Queste due caratteristiche hanno prima impedito di lavorare direttamente sul progetto esistente, poi hanno scoraggiato un porting diretto verso lazarus per la complessità di trovare le corrispondenze dirette fra l'uno e l'altro IDE.

Per questo motivo lo studio delle tessere RFID e Mifare e dei sistemi Obid®è stato operato sotto Lazarus, che in questo ambito non rappresenta una limitazione delle potenzialità per poi riportare le conoscenze verso Delphi nella fase conclusiva e realizzativa del progetto.

4.3 Usare le librerie Obid®

Come già detto più volte Obid®mette a disposizione le librerie per gli ambienti di sviluppo C++, .NET e JAVA, ma anche la possibilità di utilizzare direttamente le DLL su Delphi che riconosce e gestisce oggetti e tipi primitivi di C++. Per poter usare le DLL esse vanno inserite nella cartella del

progetto o direttamente in C:\Windows\system32, mentre le librerie .pas devono, obbligatoriamente, essere nella cartella del progetto oltre che apparire sotto uno dei campi **uses** delle unit che ne fanno uso. Di seguito verrà proposto un segmento di codice per la connessione e la comunicazione con un lettore.

Codice 4.3: Procedura per acquisizione di un UID da Tag

```
uses ... FEUSB, FEISC;

procedure TForm1.Comunicazione;
var
iDeviceHnd : integer;
iReaderHnd : integer;
stato : integer;
cBusAdr : byte;
sData : string;
cResp : array [0..255] of char;
iResp : integer;
A : integer;
FSS : FEUSB_SCANSEARCH;
begin
  //apre la connessione con il dispositivo
  iDeviceHnd := FEUSB_ScanAndOpen(FEUSB_SCAN_FIRST, FSS);

  //controlla l'avvenuta connessione
  if iDeviceHnd > 0 then
  begin
    //apre la connessione con il lettore
    iReaderHnd := FEISC_NewReader(iDeviceHnd);

    //controlla l'avvenuta connessione
    if iReaderHnd > 0 then
    begin
      //riepmpie con caratteri null la risposta
      FillChar(cResp,255,0);
      cBusAdr:=255;

      //prepara il comando: eg. inventory
      // e ne sceglie la modalita'(NON-ADDRESSED)
      sData := #$01 + #$00;

      //esegue il comando
      stato := FEISC_0xB0_ISOCmd(iReaderHnd, cBusAdr,
                   PChar(sData), Length(sData),
                   cResp, iResp, 0);

      ...

      //chiude la connessione con il lettore
```

```
      //e con il dispositivo
      FEISC_DeleteReader(iReaderHnd);
      FEUSB_CloseDevce(iDeviceHnd);

    end;
  end;
end;
```

In questo codice viene evidenziato l'ordine con cui bisogna procedere per ottenere una corretta connessione. Tuttavia, diversamente da come è mostrato nel codice, non è consigliato aprire e chiudere in continuazione connessioni specie con i dispositivi USB per i suddetti problemi di stabilità dei driver.

È quindi consigliato un approccio nel quale la connessione con il dispositivo USB avviene la prima volta con l'apertura del form e la disconnessione al momento della chiusura dello stesso form utilizzando le procedure associate agli eventi OnActivate e OnClose.

Codice 4.4: connessione in apertura e chiusura di un form

```
uses ... FEUSB, FEISC;

var
...
iDeviceHnd : integer
FSS : FEUSB_SCANSEARCH;

procedure TForm1.FormActivate(Sender: TObject);
begin
  iDeviceHnd := FEUSB_ScanAndOpen(FEUSB_SCAN_FIRST, FSS);
end;

procedure TForm1.Comunicazione;
var
...
begin
  ...
end;

procedure TForm1.FormClose(Sender: TObject;
                  var CloseAction: TCloseAction);
begin
  FEUSB_CloseDevice(iDeviceHnd);
end;
```

Nel caso fossero presenti più lettori ma si volesse aprire un canale di comunicazione con un determinato lettore, magari di una specifica famiglia, bisognerebbe avvalersi di specifiche funzioni per scansionare la lista dei lettori e trovare quello desiderato. È necessario ricordarsi che vengono aperti

molti canali con i dispositivi USB, e che quindi vanno anche tutti chiusi. Il codice 4.5 è un esempio di ricerca di un dispositivo della famiglia "OBID i-Scan Midrange".

Codice 4.5: Gestione di più lettori

```
uses ... FEUSB, FEISC;


procedure TForm1.FormActivate(Sender: TObject);
begin
  FEUSB_ScanAndOpen(FEUSB_SCAN_ALL, FSS);
end;

procedure TForm1.Comunicazione;
var
...
iNextHnd : integer;
iReaderHnd : integer;
cValue : array [0..32] of char;
begin
  iReaderHnd := 0;
  //cerca e attivo  tutti i lettori collegati tramite usb
  iNextHnd := FEUSB_GetDeviceList(0);
  //iteratore per la scansione della lista di dispositivi
  while (iNextHnd > 0) and (iReaderHnd = 0) do
  begin
     FEUSB_GetDevicePara(iNextHnd, 'FamilyName',cValue);
     if cValue = 'OBID i-scan Midrange' then
      begin
       iReaderHnd := FEISC_NewReader(iNextHnd);
      end;
     iNextHnd := FEUSB_GetDeviceList(iNextHnd);
  end;

  ...

  FEISC_DeleteReader(iReaderHnd);
end;


procedure TForm1.FormClose(Sender: TObject;
                 var CloseAction: TCloseAction);
begin
  iNextHnd := FEUSB_GetDeviceList(0);
  while iNextHnd > 0 do
  begin
     FEUSB_CloseDevice(iNextHnd);
     iNextHnd := FEUSB_GetDeviceList(iNextHnd);
  end;
```

```
end;
```

4.4 Ottenere un UID

A questo punto abbiamo le conoscenze per elaborare una unit che si colleghi contemporaneamente a più lettori di famiglie diverse, nello specifico uno della famiglia "i-scan Midrange" e uno della famiglia "Classi-pro", esegua su di essi un'istruzione di inventory e restituisca l'UID ottenuto e liberi la memoria correttamente.

Codice 4.6: unit per Ottenere un UID:intestazione della unit

```
unit Unit1;

interface

uses
  ... FEISC,FEUSB;

type
  TForm1 = class(TForm)
        procedure FormActivate(Sender : Tobject);
    procedure FormClose(Sender: TObject;
                        var CloseAction: TCloseAction);
  private

  public
    function LeggiUID:string;
  end;
var
  Form1: TForm1;
  statomr : integer;//stato del lettore middlerange
  statocpr : integer;//stato del lettore classic-pro


  cpr : integer;//Device-Handle per classic-pro USB
  mr : integer;//Device-Handle per middlerange USB
  rhcpr : integer;//Reader-Handle per classic-pro
  rhmr : integer;//Reader-Handle per middlerange

implementation
```

Nell'intestazione vediamo come ci siano alcune variabili globali per la gestione dei dispositivi e dei lettori. Esse si trovano in questa posizione perché vengono inizializzate dalla FormActivate, servono per gestire i lettori durante la lettura e poi verranno usate per liberare la memoria dagli oggetti che puntano. L'unica funzione pubblica è proprio quella per la lettura del

codice che quindi potrà essere invocata da qualsiasi classe una volta attivato il form.

Codice 4.7: unit per Ottenere un UID:connessione

```
procedure TForm1.FormActivate(Sender: TObject);
var
  //gestore temporaneo del dispositivo usb
  IDLettore: integer;
  //struttura per la ricerca su usb: da usare vuota
  FSS : FEUSB_SCANSEARCH;
  //stringa per leggere la famiglia del lettore
  cValue: array[0..255] of char;

begin
  //cerca i dispositivo sull'usb
  //e apre i canali di comunicazione
  FEUSB_ScanAndOpen(FEUSB_SCAN_ALL, FSS);
  //iteratore per la scansione della lista di dispositivi
  IDLettore := FEUSB_GetDeviceList(0);
  while IDLettore >0 do
  begin
     FEUSB_GetDevicePara(IDLettore, 'FamilyName',cValue);
     cValue[16]:=#00;
     if cValue = 'OBID classic-pro' then
     begin
       cpr := IDlettore;
     end;
     if cValue = 'OBID i-scan Midr' then
     begin
       mr := IDlettore;
     end;
     IDLettore := FEUSB_GetDeviceList(IDLettore);
  end;

  if(cpr > 0) and (mr > 0) then
    begin
      rhcpr := FEISC_NewReader(cpr);
      rhmr := FEISC_NewReader(mr);
    end
  else
    begin
      FEUSB_CloseDevice(cpr);
      FEUSB_CloseDevice(mr);
    end;
end;

procedure TForm1.FormClose(Sender: TObject;
                var CloseAction : TCloseAction);
begin
```

```
  //elimino gli oggetti associati e libero la memoria
  FEISC_DeleteReader(rhcpr);
  FEISC_DeleteReader(rhmr);
  FEUSB_CloseDevice(cpr);
  FEUSB_CloseDevice(mr);
end;

end.
```

La connessione e la disconnessione sono come le abbiamo già viste, l'unica particolarità sta nella doppia ricerca. Nel caso uno dei due lettori non sia operativo verranno chiusi entrambi. Questa gestione tiene conto del fatto che l'utente connetterà un solo lettore middle range e un solo lettore classic-pro e nessun altro lettore all'interfaccia USB.

Codice 4.8: unit per Ottenere un UID: Funzione di inventory

```
function TForm1.LeggiUID:string;
var
  bus: byte;//indirizzo del bus per mr e per cpr
  sDatacpr: string; //comandi per cpr
  cRespcpr : array [0..255] of char;// risposta per cpr
  iRespcpr: Integer;//lunghezza della risposta per cpr
  sDatamr: string; //comandi per mr
  cRespmr : array [0..255] of char;//risposta per mr
  iRespmr: Integer;//lunghezza della risposta per mr
  CodiceTag:string;//codice del tag presnte sulle antenne
  A : integer;//contatore
begin

  //mette a 0(sentinella) tutta la stringa
  FillChar(cRespcpr,255,0);
  FillChar(cRespmr,255,0);
  bus:=255;

  //cancella la memoria di eventuali letture precedenti
  FEISC_0x69_RFReset(rhcpr, buscpr);
  FEISC_0x69_RFReset(rhmr, busmr);

  //inizializzo la stringa che rappresenta l'UID
  //come se fosse per una tessera ISO-15693
  CodiceTag:='00';

  //codice per inventory modalita' NON-ADDRESSED
  sDatamr := #$01 + #$00;

  statomr := FEISC_0xB0_ISOCmd(rhmr, bus,
                  PChar(sDatamr), Length(sDatamr),
                  cRespmr, iRespmr, 0);
```

```
  //esecuzione dell'istruzione di inventory

  {se la risposta ha piu' di 11 caratteri allora c'e'
   piu' di un tag nel campo dell'antenna:troppi tag}
  If IRespmr<=11 then
    Begin
      If (Statomr>=0) and (IRespmr>0) then
        For A:=3 to 10 do
        //prendo i caratteri dal numero 3 all'ultimo
        //dove c'e' l'UID vero e proprio
          CodiceTag:=CodiceTag +
                     IntToHex(ord(Crespmr[A]),2);
    end;

  //se non ho avuto risposta dal mr procedo con il cpr
  if(CodiceTag = '')  then
  begin

    //penso ad un Tag ISO-14443
    CodiceTag := 'FF';

    sDatacpr := #$01;
    sDatacpr := sDatacpr + #$00;

    statocpr := FEISC_0xB0_ISOCmd(rhcpr, bus,
                     PChar(sDatacpr), Length(sDatacpr),
                     cRespcpr, iRespcpr, 0);

    If (Statocpr>=0) and (IRespcpr>0) then
      For A:=3 to 10 do
      //prendo i caratteri dal numero 3 all'ultimo
      //dove c'e' l'UID vero e proprio
        CodiceTag:=CodiceTag +
                   IntToHex(ord(Crespcpr[A]),2);
    else
      //nessun Tag valido sulle antenne
      CodiceTag := '';
  end;

  LeggiUID := CodiceTag;

end;
```

La funzione di lettura dell'UID decreta implicitamente una priorità: se c'è un Tag presente sull'antenna middlerange, quindi di tipo ISO-15693 allora non verrà utilizzato il lettore classic-pro. In caso contrario viene si prova a leggere l'UID da un Tag ISO-14443.

Capitolo 5

Un Esempio Realizzativo: BiblioLend

Sia l'analisi sui sistemi RFID, che quella su Delphi sono state sviluppate con un preciso obiettivo, integrare la gestione di PadovaPass in un sistema precedentemente realizzato da @bc network per l'informatizzazione e l'automazione della gestione di una biblioteca. Ciò non toglie che tutto quello che è stato detto sia utile e pensato per integrare PadovaPass con altri sistemi di erogazione servizi al cittadino.

In questo capitolo porremo attenzione proprio su Bibliolend e sulle difficoltà riscontrate per la realizzazione e l'integrazione del nostro progetto in questo già esistente. Come per la scelta di Mifare per PadovaPass, anche alcune scelte fatte dal programmatore di bibliolend ci hanno ridotto le possibilità realizzative. Nei prossimi paragrafi analizzeremo il sistema e capiremo come e dove introdurre PadovaPass

5.1 Bibliolend

Bibliolend è un sistema che permette, tramite Tag RFID ISO-15693, la gestione automatizzata di prestiti e prenotazioni di libri da parte di utenti registrati e in possesso di una tessera della biblioteca.

Questo sistema associa ad ogni Volume un UID che corrisponde a quello di un'etichetta ISO-15693 che dev'essere incollata sul libro. Anche le tessere degli utenti sono provviste di Tag ISO-15693. Gli UID di volumi e Tessere andranno a costituire le chiavi primarie nelle corrispettive tabelle del Data Base associato alla biblioteca.

L'esistenza di un Data Base, giustifica fortemente l'utilizzo di Delphi e introduce l'obbligo di connettere i lettori RFID con un DBMS residente su un PC che può essere lo stesso collegato ai lettori, o trovarsi in rete con essi. Questo fatto fa passare in secondo piano, almeno per questa possibile realizzazione, le difficoltà riscontrate nel paragrafo 3.2.3.

Naturalmente, se il DBMS lo consente, potranno esserci più terminali d'accesso messi in rete con il server dove risiede il Data Base.

L'accesso di un utente al sistema avviene avvicinando la propria tessera personale al lettore, il sistema, che in polling controlla l'antenna, riscontrato un Tag nell'area attiva ne preleverà l'UID e lo userà per identificare l'utente. Entrati nella pagina dell'utente l'antenna rileverà eventuali libri. Se essi sono già in prestito all'utente saranno da considerare in restituzione, altrimenti saranno nuovi libri che l'utente prende in prestito. Qui il programmatore ha fatto la scelta di non sfruttare una delle potenzialità più interessanti dei sistemi RFID: la gestione di Tag Multipli. Infatti con il controllo della lunghezza della risposta verifica se sul lettore sono presenti uno o più libri. nel caso ne sia presente più di uno nessun libro verrà processato.

Gli amministratori hanno maggiori diritti sul Data Base, quale quello di inserire nuovi libri, registrare nuovi utenti o bloccarne uno particolarmente ritardatario nella riconsegna dei libri. Per l'ingresso all'area riservata è necessaria la tessera associata ad un amministratore e in seguito digitare la password per quell'utente.

5.2 Integrare PadovaPass

In questo contesto vogliamo riuscire ad utilizzare le tessere in distribuzione presso la rete di APS come tessere della biblioteca in un ipotetica realizzazione di un sistema bibliotecario automatizzato per il comune o la provincia di Padova (territorio di distribuzione della tessera e di afferenza dei clienti APS).

5.2.1 Idea di partenza

L'idea iniziale era quella di sfruttare le informazioni personali sul titolare della tessera, che sono salvate all'interno di essa, per automatizzare la registrazione dell'utente. Nella nostra idea doveva bastare passare la Tessera vicino ad un antenna predisposta a questo scopo e confermare i dati prelevati per registrarsi. Questo non è stato possibile perché non possiamo accedere all'area protetta della memoria delle tessere in quanto non siamo in possesso delle chiavi previste da MiFare. Inoltre non conosciamo nello specifico il protocollo NXP che sostituisce la parte 4 del protocollo ISO-14443 (protocollo di trasmissione) per le tessere Classic.
In seguito era previsto l'uso del solo UID per l'identificazione della tessera e l'accesso al sistema. L'esistente sistema con Tag ISO-15693 resterebbe inalterato.

Di fatto questo semplifica in parte il lavoro perché non ci obbliga ad aggiornare la gestione del Data Base e creare nuovi form per queste operazioni, basterà aggiornare l'hardware aggiungendo il lettore per MiFare, e il software per il prelievo dell'UID

5.2.2 Aggiornamento Hardware e Software

L'impossibilità di realizzare l'idea sopra descritta porta quindi a dover prelevare il solo UID e associarlo nel Data Base all'insieme delle informazioni inserite manualmente.
Per integrare il sistema MiFare è inoltre, come già ampiamente spiegato nei capitoli precedenti, un aggiornamento dell'hardware e di conseguenza del software che lo gestisce.

Per quanto riguarda la parte Hardware è necessario installare lettori per entrambi i tipi di Tag. Nel capitolo 2 viene analizzato profondamente il problema e proposte delle soluzioni ai problemi riscontrati.

Per la gestione dell'hardware è necessario rivedere in parte il software. Nel sistema funzionante fornitoci da @bc network, dopo un attenta analisi delle varie unit e delle dipendenze fra di loro, abbiamo individuato in MOD.pas la unit che gestisce il prelievo dell'UID. In particolare la gestione delle operazioni con il dispositivo è affidata alla classe TDM, di cui la unit crea un esemplare di nome DM. Nelle procedure `DataModuleCreate` e `DataModuledestroy`, corrispettivi di costruttore e distruttore in C++, si trova la connessione e la disconnessione con il dispositivo USB.
Il prelievo dell'UID avvine tramite la funzione pubblica `LetturaCodice` che restituisce una stringa contenete il codice identificativo della tessera sul lettore, attivata a sua volta da una funzione a timer che si preoccupa di controllare se il Tag corrisponde a un libro o a un utente e poi compiere le operazioni adatte a quel Tag.

In questi 3 metodi si concentra il nostro sforzo per aggiornare il software all'utilizzo di MiFare. La soluzione corrisponde ai blocchi di codice 4.6, 4.7e 4.8 avendo cura di inserirli all'interno delle suddette funzioni modificandone solo la parte interessata e lasciando inalterate le altre operazioni tipo connessione al Data Base o avvio del timer.

Come spiegato nel paragrafo 4.4 nella funzione di lettura viene implicitamente introdotta una priorità di gestione dei lettori.

5.2.3 Aggiornamento Data Base

In riferimento ai problemi di discernimento fra i due tipi di Tag presentati nel paragrafo 3.2.2, si è scelto di aggiungere un Byte di controllo alla stringa che rappresenta l'identificativo. Tale Byte sarà posto a 0x00 se si tratta di una vicinity card, mentre a 0xFF se si tratta di una MiFare. Tale scelta porta a dover aggiornare il Data Base se già esistente aggiungendo due cifre esadecimali 0 (l'UID è espresso in cifra esadecimali) davanti a tutti i codici associati a libri e utenti. Se il Data Base non esiste ciò avverrà automaticamente.

Capitolo 6

Conclusioni

L'idea di utilizzare PadovaPass come semplice tessera identificativa è risultata vincente in quanto non richiede un aggiornamento del sistema troppo impegnativo. L'ampia distribuzione della tessera e la potenza degli strumenti, che ne stanno alla base, potrebbero determinare il successo di tale soluzione.

È bene notare però che se si avesse accesso alle chiavi di sicurezza e si conoscesse il protocollo proprietario NXP, che sostituisce la parte 4 del protocollo ISO-14443 modalità A, l'integrazione sarebbe completa e molto più potente al costo di un aggiornamento software più complesso. Tale soluzione utilizzerebbe a pieno le potenzialità di MiFare.

Altro ambito di applicazione già perseguito da APS Holding è l'utilizzo di PadovaPass per i trasporti di altre società e la bigliettazione di eventi. Queste due soluzioni sono perseguibili utilizzando i sistemi già sviluppati per l'obliterazione e ricarica dei titoli di viaggio APS Mobilità.

Uno sviluppo futuro del progetto si dovrà avvalere o delle chiavi di cifratura o per lo meno di tessere MiFare Plus non predisposte per PadovaPass per poter testare le funzioni protette da crittografia e completare lo studio per poter realizzare quanto pensato inizialmente.

Tavola degli Acronimi

ADO ..... ActiveX Data Objects
APS ..... Azienda Padova Servizi
ASCII ..... American Standard Code for Information Interchange
DBMS ..... Data Base Managment System
DLL ..... Dynamic-Link Library
GPL ..... General Public License
IDE ..... Integrated Development Enviroment
ISO ..... International Standard Organization
OOP ..... Object Oriented Programming
RFID ..... Radio Frequency IDentification
SDK ..... Software Development Kit
UID ..... Unique Identification Number
USB ..... Universal Serial Bus

Bibliografia

[1] *www.apsholding.it*

[2] *www.padovanet.it*

[3] *www.ne-t.it*

[4] *it.wikipedia.org*

[5] *en.wikipedia.org*

[6] *www.abcnetwork.it*

[7] *www.mifare.net*

[8] *www.nxp.com*

[9] Feig. *ID FEUSB: Software-Support for USB*. Version 3.03.04.

[10] Feig. *ID FEISC: Software-Support for Obid® i-scan and Obid® classic-pro*. Version 5.07.05.

[11] *www.aimglobal.org*

[12] *www.openpcd.org*

[13] *rfidshop.com.hk*

[14] *www.lazarus.freepascal.org*

[15] *wiki.lazarus.freepascal.org*

[16] *www.iso.com*

[17] *www.feig.de*